Laboratory 3: Calibration, Odometry and Lidar Mapping

Drini Kerciku — 1004750780
Tian Yu — 1004750780
Ben Natra — 1005117366
Smit Patel — 1004734497

March 15, 2023

1 Report

1.1 Vehicle Calibration

Q: How did you drive the robot to determine the wheel radius? How much did it rotate? How far did you drive forward?
A: To determine the wheel radius, we drove the robot straight forward and used the formulas derived in lecture. We drove forward 1m in our case, and did not rotate.

Q: How did you drive the robot to determine the wheel separation? How much did you rotate? How far did you drive forward?
A: To determine wheel separation we rotated the robot on the spot, and again used the formula derived in lecture. We rotated the robot for 3 full turns and did not drive forward at all.

Q: How does your code work or should work? How are the parameters determined?
A:The code works by using the encoder measurements on both wheels. The encoder measurements are accumulated starting when the robot starts moving. When the robot is detected to no longer be moving, the overall change in encoder measurements from the left and right wheel are used in the formulas provided in lecture to arrive at the robot parameters. These formulas are obtained by rearranging the kinematic unicycle robot model.

Q: What values did you get for the wheel radius and baseline?
A:

1. $r_{wheel} = 0.03309m$
2. $b = 0.1489$ and $2b = 0.2978$

Q: Does these values match those given in the data sheet? Identify one possible source of uncertainty or bias that made your answer differ from the factory calibration.
A:These values are fairly close to those provided by the datasheet. One source of error could be that the distance traveled and rotations performed were measured by hand, and thus, not exact. This uncertainty would add an element of randomness to the main input (the distance traveled or number of rotations). It could be mitigated by repeating the experiments many times and taking an average of the results to reduce the overall standard deviation of the measurement.

1.2 Motion Estimation

Q: How is your estimation compared to the onboard one? Name one possible source of error that accounts for the differences.
A: Our estimation is fairly similar to the onboard, however a deviation between the two values is observed over time. One explanation for this would be that the onboard odom topic uses IMU measurements in conjunction with encoder readings, and this mix of measurements would make it more robust to encoder errors than our algorithm which uses encoder measurements alone. For example, the IMU should allow for some limited correction from wheel slip since the robots actual movement will be accounted for, while the encoders have no way to determine if slip is occurring.

Q: How are the estimates compared to the actual trajectory you observed? Name one possible source of error that accounts for the differences.
A: The estimates are fairly close to the real life trajectories, although without exactly measuring the real life position it is difficult to determine the level of difference between them. For the circle, the robot roughly returned to the starting location, and for the complex shape a similar path was observed as the recording. A source of error could include wheel slip which would add error to encoder measurements, typically making them overestimate values. This is mitigated in the onboard odom topic by IMU measurements, but significant slip, caused by larger control inputs, would still cause errors when the IMU and encoder values are reconciled.

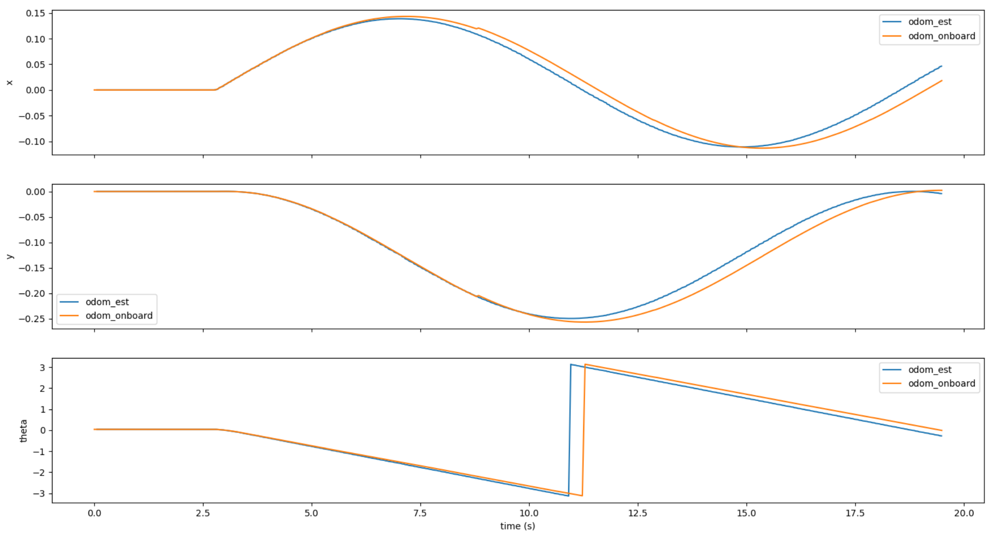

Figure 1: Motion estimation for circular motion.

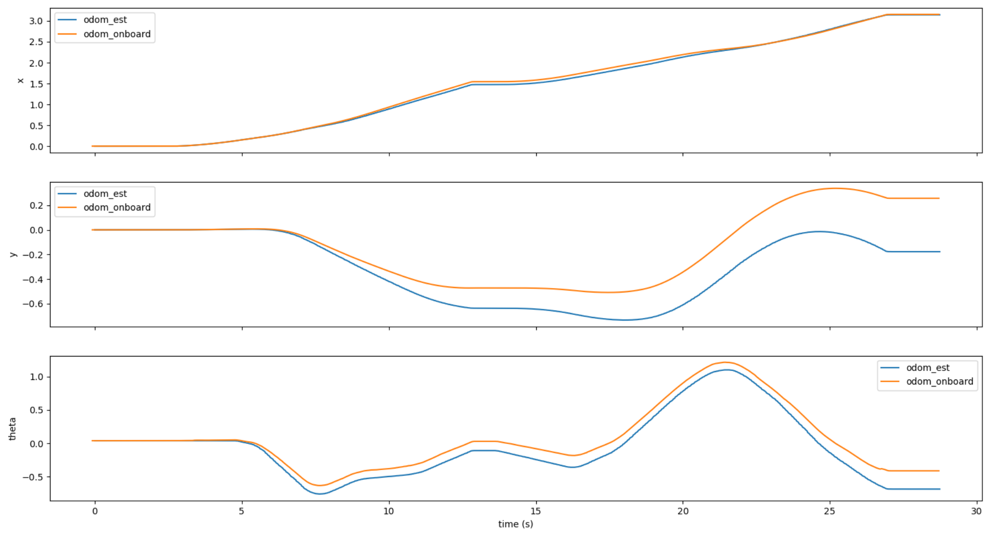

Figure 2: Motion estimation for variable controls over time.

1.3 Occupancy Grid Mapping

Q: Describe how your code works or should work.
A: The code works by using each lidar measurement to add to the log likelihood of the occupancy grid being occupied or unoccupied. At each measurement interval, all the lidar points are retrieved. They are then iterated through, and for each point the map is updated by adding "alpha" to the log likelihood of a predetermined number of points at the measurement coordinates (representing the part of the obstacle seen), and subtracting "beta" from the cells between the obstacle and the robot. A saturation value is imposed on the log likelihoods to prevent overflow errors. As the robot drives around this process is repeated to build a map of the environment

Q: Explain a potential source of error in this mapping algorithm.
A: A source of error in the algorithm could originate due to imprecision in the robots dead reckoning. Dead reckoning can be subject to accumulated errors without other measurements to reference against. This would result in an imprecise estimate of the robot location, and thus, the map would "shift" over time. As the robot's odom estimate is similar to its simulated counterpart, this error did not present itself significantly unless significant control inputs were tested. Another potential source of error would be in determining the number of points near the lidar measurements to assign as the obstacle. In the current algorithm, this is an arbitrary number of 3 points, which reduces the precision of the mapped obstacle.

Q: Discuss the results. Explain a potential source of error that did not present in simulation.
A:Overall, the mapping of the environment reflected in general the that which was setup. However, it could be noticed in the map of the environment that some of the obstacles seem to be mapped in two slightly offset locations, which results from them being mapped initially, and mapped again when the robot returns to position. This results from errors in the robot's dead reckoning for its position estimate. As mentioned in part 2, there are more factors affecting the robots position in real life than in simulation, and thus, the position estimate in real life is likely to be less accurate. Therefore, this error would be more significant in real life than in the simulation. Other errors could include moving objects not benign accounted for properly, or the lidar beam not reflecting ideally off surfaces, but these did not present themselves in any significant way during real life testing.

Figure 3: Simulation mapping of the Gazebo world.

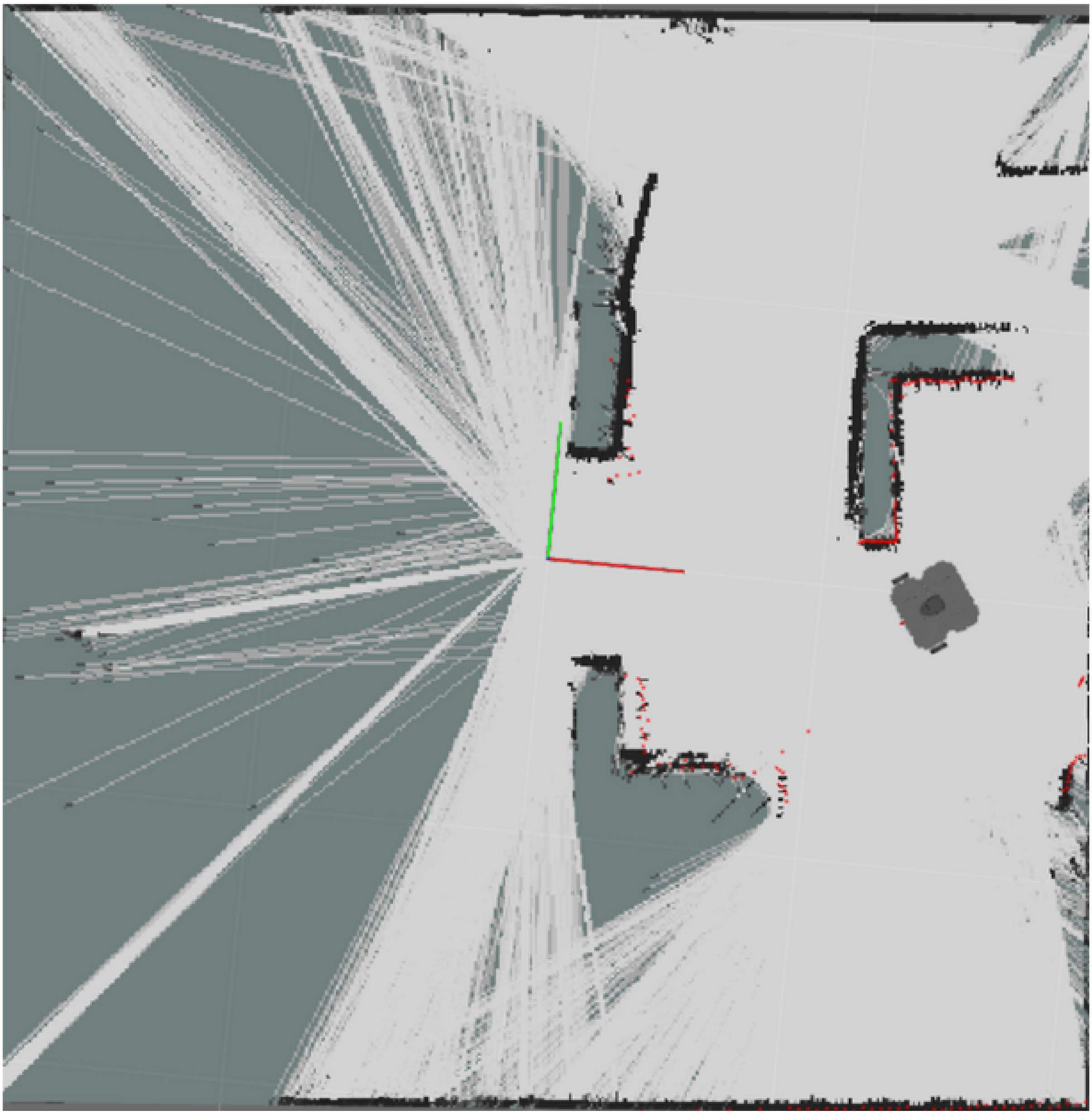

Figure 4: Mapping of the physical maze in the laboratory.

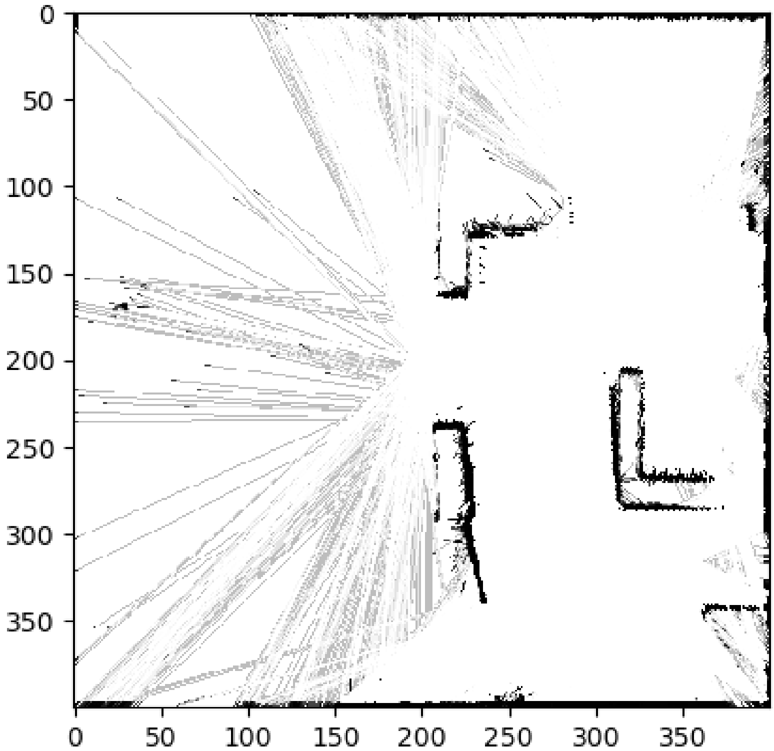

Figure 5: Obstacle map in the laboratory.

2 Source Code

2.1 Wheel Radius Estimation : l3_estimate_wheel_radius.py

```
#!/usr/bin/env python3

import rospy
import numpy as np
import threading
from turtlebot3_msgs.msg import SensorState
from nav_msgs.msg import Odometry
from std_msgs.msg import Empty
from geometry_msgs.msg import Twist

from utils import convert_pose_to_tf, euler_from_ros_quat, ros_quat_from_euler


INT32_MAX = 2**31
DRIVEN_DISTANCE = 0.75 #in meters
TICKS_PER_ROTATION = 4096

class wheelRadiusEstimator():
    def __init__(self):
        rospy.init_node('encoder_data', anonymous=True) # Initialize node

        #Subscriber bank
        rospy.Subscriber("cmd_vel", Twist, self.startStopCallback)
        rospy.Subscriber("sensor_state", SensorState, self.sensorCallback) #Subscribe to the sensor state msg

        #Publisher bank
        self.reset_pub = rospy.Publisher('reset', Empty, queue_size=1)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, self.odom_cb, queue_size=1)

        #Initialize variables
        self.odom = Odometry()
        self.left_encoder_prev = None
        self.right_encoder_prev = None
        self.del_left_encoder = 0
        self.del_right_encoder = 0
        self.isMoving = False #Moving or not moving
        self.lock = threading.Lock()

        #Reset the robot
        reset_msg = Empty()
        self.reset_pub.publish(reset_msg)
        print('Ready to start wheel radius calibration!')
        return

    def safeDelPhi(self, a, b):
        #Need to check if the encoder storage variable has overflowed
        diff = np.int64(b) - np.int64(a)
        if diff < -np.int64(INT32_MAX): #Overflowed
            delPhi = (INT32_MAX - 1 - a) + (INT32_MAX + b) + 1
        elif diff > np.int64(INT32_MAX) - 1: #Underflowed
            delPhi = (INT32_MAX + a) + (INT32_MAX - 1 - b) + 1
        else:
            delPhi = b - a
        return delPhi

    def sensorCallback(self, msg):
        #Retrieve the encoder data form the sensor state msg
        self.lock.acquire()
        if self.left_encoder_prev is None or self.left_encoder_prev is None:
            self.left_encoder_prev = msg.left_encoder #int32
            self.right_encoder_prev = msg.right_encoder #int32
        else:
            #Calculate and integrate the change in encoder value
            #Compares current encoder value with previous one. If curret one is an overflow or underflow, deal with it. Accums in del_left/right_encoder
            self.del_left_encoder += self.safeDelPhi(self.left_encoder_prev, msg.left_encoder)
            self.del_right_encoder += self.safeDelPhi(self.right_encoder_prev, msg.
```

```
    right_encoder)

            #Store the new encoder values
            self.left_encoder_prev = msg.left_encoder #int32
            self.right_encoder_prev = msg.right_encoder #int32
        self.lock.release()
        return

    def startStopCallback(self, msg):
        input_velocity_mag = np.linalg.norm(np.array([msg.linear.x, msg.linear.y, msg.linear.z]))
        if self.isMoving is False and np.absolute(input_velocity_mag) > 0:
            self.isMoving = True #Set state to moving
            print('Starting Calibration Procedure')

        elif self.isMoving is True and np.isclose(input_velocity_mag, 0):
            self.isMoving = False #Set the state to stopped

            # # YOUR CODE HERE!!!
            # Calculate the radius of the wheel based on encoder measurements

            # get radians travelled from left and right wheel
            left_rads = self.del_left_encoder/TICKS_PER_ROTATION * 2 * np.pi
            right_rads = self.del_right_encoder/TICKS_PER_ROTATION * 2 * np.pi

            #get wheel radius
            radius = 2*DRIVEN_DISTANCE/(left_rads + right_rads)

            print('Calibrated Radius: {} m'.format(radius))

            #Reset the robot and calibration routine
            self.lock.acquire()
            self.left_encoder_prev = None
            self.right_encoder_prev = None
            self.del_left_encoder = 0
            self.del_right_encoder = 0
            self.lock.release()
            reset_msg = Empty()
            self.reset_pub.publish(reset_msg)
            print('Resetted the robot to calibrate again!')

        return

    def odom_cb(self, odom_msg):
        # get odom from turtlebot3 packages
        self.odom = odom_msg
        print("Turtlebot3 Odom: x: %2.3f, y: %2.3f, theta: %2.3f" % (
            self.odom.pose.pose.position.x, self.odom.pose.pose.position.y,
            euler_from_ros_quat(self.odom.pose.pose.orientation)[2]
            ))

        # self.bag.write('odom_onboard', self.odom)


if __name__ == '__main__':
    Estimator = wheelRadiusEstimator() #create instance
    rospy.spin()
```

2.2 Wheel Base Estimation : l3_estimate_wheel_baseline.py

```
#!/usr/bin/env python3

import rospy
import numpy as np
import threading
from turtlebot3_msgs.msg import SensorState
from std_msgs.msg import Empty
from geometry_msgs.msg import Twist

INT32_MAX = 2**31
NUM_ROTATIONS = 3
TICKS_PER_ROTATION = 4096
WHEEL_RADIUS = 0.066 / 2 #In meters
```

```python


class wheelBaselineEstimator():
    def __init__(self):
        rospy.init_node('encoder_data', anonymous=True) # Initialize node

        #Subscriber bank
        rospy.Subscriber("cmd_vel", Twist, self.startStopCallback)
        rospy.Subscriber("sensor_state", SensorState, self.sensorCallback) #Subscribe to the sensor state msg

        #Publisher bank
        self.reset_pub = rospy.Publisher('reset', Empty, queue_size=1)

        #Initialize variables
        self.left_encoder_prev = None
        self.right_encoder_prev = None
        self.del_left_encoder = 0
        self.del_right_encoder = 0
        self.isMoving = False #Moving or not moving
        self.lock = threading.Lock()

        #Reset the robot
        reset_msg = Empty()
        self.reset_pub.publish(reset_msg)
        print('Ready to start wheel radius calibration!')
        return

    def safeDelPhi(self, a, b):
        #Need to check if the encoder storage variable has overflowed
        diff = np.int64(b) - np.int64(a)
        if diff < -np.int64(INT32_MAX): #Overflowed
            delPhi = (INT32_MAX - 1 - a) + (INT32_MAX + b) + 1
        elif diff > np.int64(INT32_MAX) - 1: #Underflowed
            delPhi = (INT32_MAX + a) + (INT32_MAX - 1 - b) + 1
        else:
            delPhi = b - a
        return delPhi

    def sensorCallback(self, msg):
        #Retrieve the encoder data form the sensor state msg
        self.lock.acquire()
        if self.left_encoder_prev is None or self.left_encoder_prev is None:
            self.left_encoder_prev = msg.left_encoder #int32
            self.right_encoder_prev = msg.right_encoder #int32
        else:
            #Calculate and integrate the change in encoder value
            self.del_left_encoder += self.safeDelPhi(self.left_encoder_prev, msg.left_encoder)
            self.del_right_encoder += self.safeDelPhi(self.right_encoder_prev, msg.right_encoder)

            #Store the new encoder values
            self.left_encoder_prev = msg.left_encoder #int32
            self.right_encoder_prev = msg.right_encoder #int32
        self.lock.release()
        return

    def startStopCallback(self, msg):

        print("is moving: {}, current vel, {}".format(self.isMoving, msg.angular.z))

        if self.isMoving is False and np.absolute(msg.angular.z) > 0:
            self.isMoving = True #Set state to moving
            print('Starting Calibration Procedure')

        elif self.isMoving is True and np.isclose(msg.angular.z, 0):
            self.isMoving = False #Set the state to stopped

            # # YOUR CODE HERE!!!
            # Calculate the radius of the wheel based on encoder measurements

            # get radians travelled from left and right wheel
```

```
            left_rads = self.del_left_encoder/TICKS_PER_ROTATION * 2 * np.pi
            right_rads = self.del_right_encoder/TICKS_PER_ROTATION * 2 * np.pi

            separation = WHEEL_RADIUS/2 * (right_rads - left_rads) / (NUM_ROTATIONS * 2 * np.pi)
            print('Calibrated Separation: {} m'.format(separation))

            #Reset the robot and calibration routine
            self.lock.acquire()
            self.left_encoder_prev = None
            self.right_encoder_prev = None
            self.del_left_encoder = 0
            self.del_right_encoder = 0
            self.lock.release()
            reset_msg = Empty()
            self.reset_pub.publish(reset_msg)
            print('Resetted the robot to calibrate again!')

        return


if __name__ == '__main__':
    Estimator = wheelBaselineEstimator() #create instance
    rospy.spin()
```

2.3 Robot Motion Estimation : l3_estimate_robot_motion.py

```
#!/usr/bin/env python3
from __future__ import division, print_function
import time

import numpy as np
import rospy
import tf_conversions
import tf2_ros
import rosbag
import rospkg

# msgs
from turtlebot3_msgs.msg import SensorState
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Pose, Twist, TransformStamped, Transform, Quaternion
from std_msgs.msg import Empty

from utils import convert_pose_to_tf, euler_from_ros_quat, ros_quat_from_euler

INT32_MAX = 2**31
ENC_TICKS = 4096
RAD_PER_TICK = 0.001533981
WHEEL_RADIUS = .066 / 2
BASELINE = .287/2


PHI_COL = lambda leftE, rightE : np.array([[rightE], [leftE]])
G_Q_MATRIX = np.array([[WHEEL_RADIUS/2 , WHEEL_RADIUS/2],
                       [WHEEL_RADIUS/(2*BASELINE), -WHEEL_RADIUS/(2*BASELINE)]])
B_Q = lambda robHead : np.array([[np.cos(robHead), 0],
                                 [np.sin(robHead), 0],
                                 [0, 1]])

class WheelOdom:
    def __init__(self):
        # publishers, subscribers, tf broadcaster
        self.sensor_state_sub = rospy.Subscriber('/sensor_state', SensorState, self.sensor_state_cb, queue_size=1)
        self.odom_sub = rospy.Subscriber('/odom', Odometry, self.odom_cb, queue_size=1)
        self.wheel_odom_pub = rospy.Publisher('/wheel_odom', Odometry, queue_size=1)
        self.tf_br = tf2_ros.TransformBroadcaster()

        # attributes
        self.odom = Odometry()
        self.odom.pose.pose.position.x = 1e10
        self.wheel_odom = Odometry()
```

```
        self.wheel_odom.header.frame_id = 'odom'
        self.wheel_odom.child_frame_id = 'wo_base_link'
        self.wheel_odom_tf = TransformStamped()
        self.wheel_odom_tf.header.frame_id = 'odom'
        self.wheel_odom_tf.child_frame_id = 'wo_base_link'
        self.pose = Pose()
        self.pose.orientation.w = 1.0
        self.twist = Twist()
        self.last_enc_l = None
        self.last_enc_r = None
        self.last_time = None

        # rosbag
        rospack = rospkg.RosPack()
        path = rospack.get_path("rob521_lab3")
        self.bag = rosbag.Bag(path+"/motion_estimate.bag", 'w')

        self.here = False

        # reset current odometry to allow comparison with this node
        reset_pub = rospy.Publisher('/reset', Empty, queue_size=1, latch=True)
        reset_pub.publish(Empty())
        while not rospy.is_shutdown() and (self.odom.pose.pose.position.x >= 1e-3 or
    self.odom.pose.pose.position.y >= 1e-3 or
                self.odom.pose.pose.orientation.z >= 1e-2):
            time.sleep(0.2)  # allow reset_pub to be ready to publish
        print('Robot odometry reset.')

        rospy.spin()
        self.bag.close()
        print("saving bag")

    def sensor_state_cb(self, sensor_state_msg):
        # Callback for whenever a new encoder message is published
        # set initial encoder pose
        if self.last_enc_l is None:
            self.last_enc_l = sensor_state_msg.left_encoder
            self.last_enc_r = sensor_state_msg.right_encoder
            self.last_time = sensor_state_msg.header.stamp
        else:
            # update calculated pose and twist with new data
            le = sensor_state_msg.left_encoder
            re = sensor_state_msg.right_encoder

            if not self.here:

                self.pose.orientation = self.odom.pose.pose.orientation
                self.here = True

            # # YOUR CODE HERE!!!
            # Update your odom estimates with the latest encoder measurements and
    populate the relevant area
            # of self.pose and self.twist with estimated position, heading and velocity

            # difference of ticks and time step
            diffL = self.safeDelPhi(self.last_enc_l, le)
            diffR = self.safeDelPhi(self.last_enc_r, re)

            print(le, re)

            dt = (sensor_state_msg.header.stamp- self.last_time).to_sec()

            # arc angle from the encoder
            dotPhi_L = 2*np.pi*diffL/ENC_TICKS
            dotPhi_R = 2*np.pi*diffR/ENC_TICKS

            # compute velocities
            currQuart = self.pose.orientation
            # get currHead
            _, _, currHead = euler_from_ros_quat(currQuart)
            print(currHead)

            thetaB = B_Q(currHead)
```

```python
            enc = PHI_COL(dotPhi_R, dotPhi_L)
            dotQ = np.dot(thetaB, np.dot(G_Q_MATRIX, enc))

            # save new pose
            self.pose.position.x += dotQ[0]
            self.pose.position.y += dotQ[1]
            # deal with quartenion
            currAngle = currHead - dotQ[2]
            newQ = ros_quat_from_euler([0, 0, currAngle])
            self.pose.orientation = newQ

            self.twist.linear.x = dotQ[0]/dt
            self.twist.linear.y = dotQ[1]/dt
            self.twist.angular.z = dotQ[2]/dt

            # publish the updates as a topic and in the tf tree
            current_time = rospy.Time.now()
            self.wheel_odom_tf.header.stamp = current_time
            self.wheel_odom_tf.transform = convert_pose_to_tf(self.pose)
            self.tf_br.sendTransform(self.wheel_odom_tf)

            self.wheel_odom.header.stamp = current_time
            self.wheel_odom.pose.pose = self.pose
            self.wheel_odom.twist.twist = self.twist
            self.wheel_odom_pub.publish(self.wheel_odom)

            # save the current reading to use for next callback
            self.last_enc_l = sensor_state_msg.left_encoder
            self.last_enc_r = sensor_state_msg.right_encoder
            self.last_time = sensor_state_msg.header.stamp

            self.bag.write('odom_est', self.wheel_odom)
            self.bag.write('odom_onboard', self.odom)

            # for testing against actual odom
            print("Wheel Odom: x: %2.3f, y: %2.3f, theta: %2.3f, dt: %2.3f" % (
                self.pose.position.x, self.pose.position.y, euler_from_ros_quat(self.pose.orientation)[2], dt
            ))

            print("Turtlebot3 Odom: x: %2.3f, y: %2.3f, theta: %2.3f" % (
                self.odom.pose.pose.position.x, self.odom.pose.pose.position.y,
                euler_from_ros_quat(self.odom.pose.pose.orientation)[2]
            ))

    def safeDelPhi(self, a, b):
        #Need to check if the encoder storage variable has overflowed
        diff = np.int64(b) - np.int64(a)
        if diff < -np.int64(INT32_MAX): #Overflowed
            delPhi = (INT32_MAX - 1 - a) + (INT32_MAX + b) + 1
        elif diff > np.int64(INT32_MAX) - 1: #Underflowed
            delPhi = (INT32_MAX + a) + (INT32_MAX - 1 - b) + 1
        else:
            delPhi = b - a
        return delPhi

    def odom_cb(self, odom_msg):
        # get odom from turtlebot3 packages
        self.odom = odom_msg

    def plot(self, bag):
        data = {"odom_est":{"time":[], "data":[]},
                "odom_onboard":{'time':[], "data":[]}}
        for topic, msg, t in bag.read_messages(topics=['odom_est', 'odom_onboard']):
            print(msg)


if __name__ == '__main__':
    try:
        rospy.init_node('wheel_odometry')
        wheel_odom = WheelOdom()
    except rospy.ROSInterruptException:
        pass
```

2.4 Mapping : l3_mapping.py

```python
#!/usr/bin/env python3
from __future__ import division, print_function

import numpy as np
import rospy
import tf2_ros
from skimage.draw import line as ray_trace
import rospkg
import matplotlib.pyplot as plt

# msgs
from nav_msgs.msg import OccupancyGrid, MapMetaData
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import LaserScan

from utils import convert_pose_to_tf, convert_tf_to_pose, euler_from_ros_quat, \
     tf_to_tf_mat, tf_mat_to_tf


ALPHA = 1
BETA = 1
MAP_DIM = (4, 4)
CELL_SIZE = .01
NUM_PTS_OBSTACLE = 3
SCAN_DOWNSAMPLE = 5

class OccupancyGripMap:
    def __init__(self):
        # use tf2 buffer to access transforms between existing frames in tf tree
        self.tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self.tf_buffer)
        self.tf_br = tf2_ros.TransformBroadcaster()

        # subscribers and publishers
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, self.scan_cb, queue_size=1)
        self.map_pub = rospy.Publisher('/map', OccupancyGrid, queue_size=1)

        # attributes
        width = int(MAP_DIM[0] / CELL_SIZE); height = int(MAP_DIM[1] / CELL_SIZE)
        self.log_odds = np.zeros((width, height))
        self.np_map = np.ones((width, height), dtype=np.uint8) * -1  # -1 for unknown
        self.map_msg = OccupancyGrid()
        self.map_msg.info = MapMetaData()
        self.map_msg.info.resolution = CELL_SIZE
        self.map_msg.info.width = width
        self.map_msg.info.height = height

        # transforms
        self.base_link_scan_tf = self.tf_buffer.lookup_transform('base_link', 'base_scan', rospy.Time(0),
                                                                  rospy.Duration(2.0))
        odom_tf = self.tf_buffer.lookup_transform('odom', 'base_link', rospy.Time(0), rospy.Duration(2.0)).transform

        # set origin to center of map
        rob_to_mid_origin_tf_mat = np.eye(4)
        rob_to_mid_origin_tf_mat[0, 3] = -width / 2 * CELL_SIZE
        rob_to_mid_origin_tf_mat[1, 3] = -height / 2 * CELL_SIZE
        odom_tf_mat = tf_to_tf_mat(odom_tf)
        self.map_msg.info.origin = convert_tf_to_pose(tf_mat_to_tf(odom_tf_mat.dot(rob_to_mid_origin_tf_mat)))

        # map to odom broadcaster
        self.map_odom_timer = rospy.Timer(rospy.Duration(0.1), self.broadcast_map_odom)
        self.map_odom_tf = TransformStamped()
        self.map_odom_tf.header.frame_id = 'map'
        self.map_odom_tf.child_frame_id = 'odom'
        self.map_odom_tf.transform.rotation.w = 1.0

        rospy.spin()
        plt.imshow(100-self.np_map, cmap='gray', vmin=0, vmax=100)
```

```python
        rospack = rospkg.RosPack()
        path = rospack.get_path("rob521_lab3")
        plt.savefig(path+"/map.png")

    def broadcast_map_odom(self, e):
        self.map_odom_tf.header.stamp = rospy.Time.now()
        self.tf_br.sendTransform(self.map_odom_tf)

    def scan_cb(self, scan_msg):
        # read new laser data and populate map
        # get current odometry robot pose
        try:
            odom_tf = self.tf_buffer.lookup_transform('odom', 'base_scan', rospy.Time(0)).transform
        except tf2_ros.TransformException:
            rospy.logwarn('Pose from odom lookup failed. Using origin as odom.')
            odom_tf = convert_pose_to_tf(self.map_msg.info.origin)

        # get odom in frame of map
        odom_map_tf = tf_mat_to_tf(
            np.linalg.inv(tf_to_tf_mat(convert_pose_to_tf(self.map_msg.info.origin))).dot(tf_to_tf_mat(odom_tf))
        )
        odom_map = np.zeros(3)
        odom_map[0] = odom_map_tf.translation.x
        odom_map[1] = odom_map_tf.translation.y
        odom_map[2] = euler_from_ros_quat(odom_map_tf.rotation)[2]

        # print(scan_msg.angle_min)
        # print(scan_msg.angle_increment)
        # print(len(scan_msg.ranges))

        grid_angle = odom_map[2] + scan_msg.angle_min
        # for i in range(0, 10):
        #     print("{:.2f} ".format(scan_msg.ranges[i]), end = "")
        # print("")
        for i, dist in enumerate(scan_msg.ranges):
            # print(grid_angle)
            if i % SCAN_DOWNSAMPLE == 0:
                if dist < scan_msg.range_max:
                    x_grid = int(odom_map[0]/CELL_SIZE)
                    y_grid = int(odom_map[1]/CELL_SIZE)
                    self.np_map, self.log_odds = self.ray_trace_update(self.np_map, self.log_odds, x_grid, y_grid, grid_angle, dist)
            # if i > 10:
            #     break
            grid_angle = grid_angle + scan_msg.angle_increment
        # print(scan_msg.angle_max)
        # print(grid_angle)
        # print("")

        # YOUR CODE HERE!!! Loop through each measurement in scan_msg to get the correct angle and
        # x_start and y_start to send to your ray_trace_update function.

        # publish the message
        self.map_msg.info.map_load_time = rospy.Time.now()
        self.map_msg.data = self.np_map.flatten()
        self.map_pub.publish(self.map_msg)

    def ray_trace_update(self, map, log_odds, x_start, y_start, angle, range_mes):
        """
        A ray tracing grid update as described in the lab document.

        :param map: The numpy map.
        :param log_odds: The map of log odds values.
        :param x_start: The x starting point in the map coordinate frame (i.e. the x 'pixel' that the robot is in).
        :param y_start: The y starting point in the map coordinate frame (i.e. the y 'pixel' that the robot is in).
        :param angle: The ray angle relative to the x axis of the map.
        :param range_mes: The range of the measurement along the ray.
        :return: The numpy map and the log odds updated along a single ray.
```

```
        """
        # YOUR CODE HERE!!! You should modify the log_odds object and the numpy map based on the outputs from
        # ray_trace and the equations from class. Your numpy map must be an array of int8s with 0 to 100 representing
        # probability of occupancy, and -1 representing unknown.
        # TODO: check for infinity

        # draw line from ray trace
        range_px = range_mes/CELL_SIZE
        x_end = int(x_start + np.cos(angle) * range_px)
        y_end = int(y_start + np.sin(angle) * range_px)
        max_y, max_x = log_odds.shape
        if x_end >= max_x:
            x_end = max_x - 1
        if y_end >= max_y:
            y_end = max_y - 1
        if x_end < 0:
            x_end = 0
        if y_end < 0:
            y_end = 0

        rr, cc = ray_trace(int(y_start), int(x_start), y_end, x_end)

        # print(log_odds.shape)
        #print("{},{}".format(x_start, y_start))
        #print("{},{}".format(x_end, y_end))
        # print("")

        log_odds[rr, cc] += -BETA
        for i in range(1, min(NUM_PTS_OBSTACLE+1, len(rr))):
            log_odds[rr[-i], cc[-i]] += BETA + ALPHA

        for i, mod_px_yval in enumerate(rr):
            mod_px_xval = cc[i]
            log_prob = log_odds[mod_px_yval, mod_px_xval]
            map[mod_px_yval, mod_px_xval] = int(100*self.log_odds_to_probability(log_prob))

        return map, log_odds

    def log_odds_to_probability(self, values):
        # print(values)
        return np.exp(values) / (1 + np.exp(values))


if __name__ == '__main__':
    try:
        rospy.init_node('mapping')
        ogm = OccupancyGripMap()
    except rospy.ROSInterruptException:
        pass
```